Aritmetica dei puntatori

Sia P una espressione di tipo `T*` e sia I una espressione intera (ossia, un'espressione il cui tipo è uno dei tipi interi).

Allora

$$P + I$$

è una espressione avente

- Tipo: `T*`
- Valore:

$$P \ + \ sizeof(T) * I$$

Essendo la somma commutativa, $P + I$ è uguale a $I + P$.

Esempio 1

```
int a
int *p = &a;    // p ha tipo int* e valore &a
```

Supponiamo che l'indirizzo della variabile a sia 1000 e `sizeof(int)` sia 4 (byte):

Allora:

- `p+1` ha tipo `int*` e valore

$$1000 + 4 * 1 \ = \ 1004$$

- `p-2` (equivalente a `p+(-2)`) ha tipo `int*` e valore

$$1000 + 4 * (-2) \ = \ 992$$

Esempio 2

```
int b
double *q = &b;    // q ha tipo int* e valore &b
```

Supponiamo che l'indirizzo della variabile b sia 2000 e `sizeof(double)` sia 8 (byte):

- `q+1` ha tipo `double*` e valore

$$2000 + 8 * 1 \ = \ 2008$$

- `q-2` ha tipo `double*` e valore

$$2000 + 8 * (-2) \ = \ 1984$$

Esempio 3

```
double  c;
double *x = &c;    // x ha tipo double*  e valore &c
double **r = &x;   // r ha tipo double** e valore &x
```

Supponiamo che `&c=500`, `&x=600`, `sizeof(double)=8` e `sizeof(double*)=4`.

- `r+1` ha tipo `double**` e valore

$$600 + 4 * 1 \ = \ 604$$

- `(*r)+1` (equivalente a `x+1`) ha tipo `double*` e valore

$$500 + 8 * 1 \ = \ 508$$

L'aritmetica dei puntatori si applica solo quando un operando ha tipo puntatore e l'altro ha tipo intero.

Esempio

```
int a;
```

Supponendo che `&a` valga 1000 e che `sizeof(int)` valga 4:

- L'espressione
  `&a + 1`
  ha tipo `int*` e valore 1004.
- L'espressione
  `(unsigned int) &a + 1`
  ha tipo `unsigned int` e valore 1001 (somma di due interi di tipo `unsigned int`).

Non è invece definita la somma fra due espressioni di tipo puntatore.

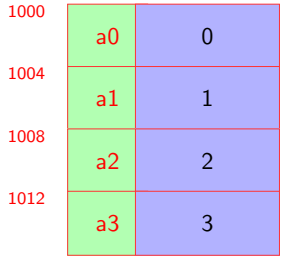
Un esempio più significativo

Date le variabili

```
int a0=0, a1=1, a2=2, a3=3;
```

supponiamo che le corrispondenti locazioni di memoria siano contigue come nella figura.

`sizeof(int) = 4`
`&a0` = 1000 , `&a1` = 1004, `&a2` = 1008, `&a3` = 1012



Espressione	Tipo	Valore	
p	int*	1000	(= &a0)
p+1	int*	1004	(= &a1)
p+2	int*	1008	(= &a2)
p+3	int*	1012	(= &a3)

Nelle ipotesi precedenti, l'assegnamento

```
*(p+1) =  -1;
```

equivale a

```
a1 =  -1;
```

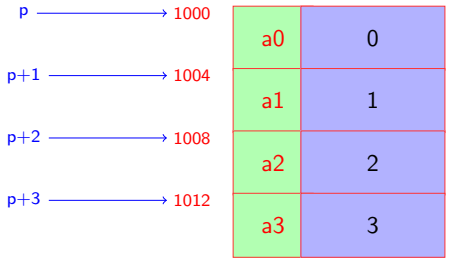
Infatti, `p+1` è l'indirizzo della locazione `a1`, quindi `*(p+1)` è sinonimo di `a1`.

Nota
Le parentesi tonde sono obbligatorie. Infatti, avendo * maggiore priorità di +, `*p+1` equivale a `(*p)+1` (espressione di tipo `int` e valore 0+1=1).

Posto

```
int* p = &a0;  // il valore di p e' 1000
```

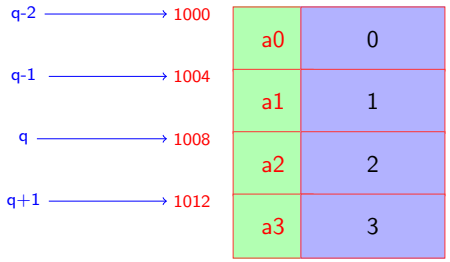
si ha:



Poniamo ora

```
int* q = &a2;  // il valore di q e' 1008
```

Allora:



Espressione	Tipo	Valore	
q-2	int*	1000	(= &a0)
q-1	int*	1004	(= &a1)
q	int*	1008	(= &a2)
q+1	int*	1012	(= &a3)

Quindi:

- L'assegnamento
  `*(q+1) = 20;`
  equivale a
  `a3 = 20;`
- L'assegnamento
  `*(q-1) = 30;`
  equivale a
  `a1 = 30;`

L'operatore []

Sia P una espressione di tipo `T*` e sia I una espressione intera. Allora l'espressione

$$P[I]$$

costruita applicando l'operatore [] alle espressioni P ed I è equivalente all'espressione

$$*(P + I)$$

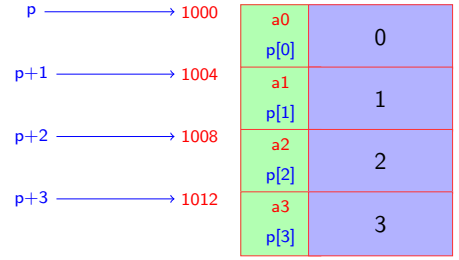
Gli assegnamenti

```
*(p+1) = -1;      *(q+1) = 20;          *(q-1) = 30;
```

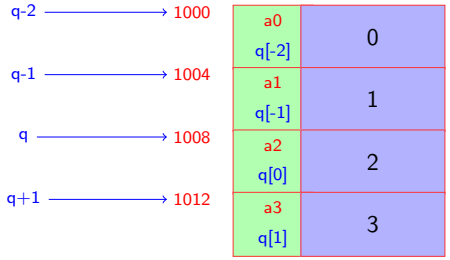
degli esempi precedenti possono essere riscritti come

```
  p[1] = -1;        q[1] = 20;           q[-1] = 30;
```


In altri termini, `p[0]` (= `*(p+0)` = `*p`) è un altro nome per la locazione `a0`, `p[1]` è sinonimo di `a1`, ecc.



Analogamente:



L'esempio di prima si può riformulare definendo

```
double a0, a1, a2, a3;
double *p=&a0, *q=&a2;
```

ed assumendo che le variabili a0,...,a3 corrispondano a locazioni di memoria contigue.

Infatti, pur non conoscendo il valore assoluto degli indirizzi e il valore di `sizeof(double)`, l'artmetica dei puntatori stabilisce che

Espressione	Tipo	Valore
p+1	double*	&a1
q+1	double*	&a3
q-1	double*	&a1

Gli assegnamenti

```
p[1] = -1.5;      q[1] = 20.3;      q[-1] = 30.7;
```

equivalgono a

```
*(p+1) = -1.5;    *(q+1) = 20.3;    *(q-1) = 30.7;
```


Array

Negli esempi precedenti, abbiamo assunto di poter definire locazioni contigue di memoria dello stesso tipo.
In C è possibile fare questo mediante gli array.

- Sintassi della dichiarazione di un array

  <array_decl> ::= <type><nome_array>[<espr_intera>]

  dove:
  - <espr_intera> è un'espressione intera costante (ossia, calcolabile in fase di compilazione).
  - <nome_array> è un identificatore.
  - <type> è il nome di un tipo.

- Semantica

  Supponiamo che in una funzione `f()` contenga la dichiarazione

  `T a[n];`

  dove `T` è il nome di un tipo, `a` è il nome dell'array e n è un intero positivo. Allora:

  - Nel record di attivazione di `f()`, viene allocata una locazione di memoria L di dimensione

    $$sizeof(T) * n$$

  - Il nome `a` dell'array è una *costante* avente tipo `T*` e valore l'indirizzo di della locazione L.
  - Per quanto discusso prima, la locazione di memoria L può essere partizionata nelle locazioni contigue

    $$a[0],\ a[1], \ldots a[n-1]$$

    ciascuna delle quali ha dimensione `sizeof(T)`.

Esempio di dichiarazione

```
int a[4];
```

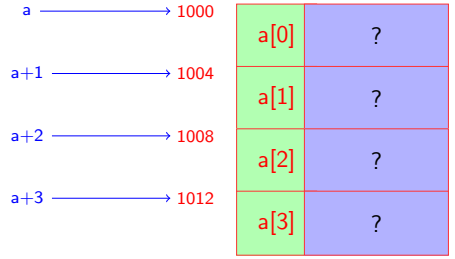
a è una costante di tipo int* il cui valore è l'indirizzo di una locazione di memoria L di dimensione sizeof(int)*4.

Avendo a tipo int*, le espressioni a+1, a+2,... hanno il significato visto con l'aritmetica dei puntatori.

Quindi, L può essere pensata come composta da quattro locazioni contigue di tipo int, i cui nomi sono rispettivamente a[0], a[1], a[2] e a[3].

Nota
Seguendo la convenzione del libro, scriviamo a[] per indicare che a è il nome di un array.

Assumendo che l'indirizzo di L sia 1000 e che sizeof(int)=4, si può rappresentare L come in figura.



Quindi, a è uguale a &a[0], a+1 è uguale a &a[1], ecc.

Il valore iniziale delle locazioni a[0],..., a[3] non è definito.

L'assegnamento

```
a[2] = 5;
```

pone nella locazione di memoria a[2] il valore 5. Equivale a:

```
*(a+2) = 5;
```

Non sono invece ammessi assegnamenti del tipo

```
int n;
...
a = &n;
```

in quanto a è una costante, quindi il suo valore non può essere modificato.

Differenze con Java

Diversamente da quanto avviene in Java, in C un array non è visto come un *oggetto* con associato uno stato.

In C, una espressione di tipo a[k] non ha alcun significato astratto, ma è solo una notazione per indicare la locazione di memoria di indirizzo a+k.

In particolare:

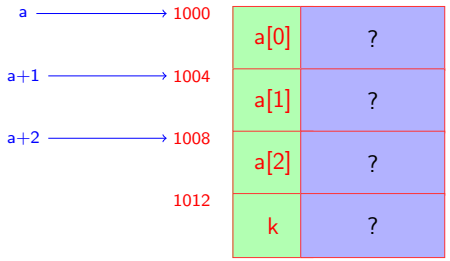
- Non c'è alcun modo per conoscere la dimensione di un array (non esiste nulla di analogo al campo length in Java).
- In esecuzione non c'è alcun controllo sugli indici di un array.

Esempio

Date le definizioni

```
int a[3]; // array di 3 elementi di tipo int
int k;
```

supponiamo che le le variabili siano disposte in memoria come nella figura



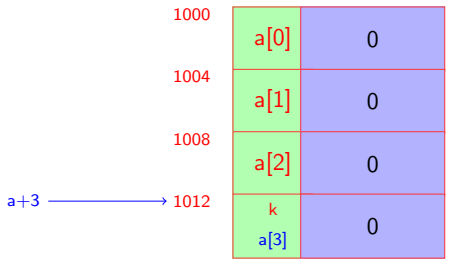
Deve essere ore eseguito l'assegnamento

```
a[3] = 3;
```

equivalente a

```
*(a+3) = 3;
```

Ma il *valore* di `a+3` è l'indirizzo della variabile `k`, quindi `a[3]` è un alias di `k`. L'assegnamento ha come effetto quello di porre $k = 0$.



Di conseguenza il ciclo non termina mai.

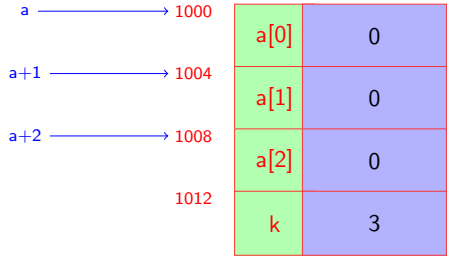
Proviamo ad eseguire il ciclo

```
for(k=0 ; k <= 3 ; k++)
    a[k] = 0;
```

Il ciclo è scorreto in quanto l'ultimo indice dell'array è 2 e non 3.

- Alla prima iterazione, viene posto $k = 0$, $a[0] = 0$ e k diventa 1.
- Alla seconda iterazione, viene posto $a[1] = 0$ e k diventa 2.
- Alla terza iterazione, viene posto $a[2] = 0$ e k diventa 3.

Al termine della terza iterazione si ha:



Nota

In esecuzione *non* viene segnalato alcun avvertimento in seguito a un errato accesso ad array (diversamente da quanto accade in Java o in altri linguaggi di alto livello).

Programmi di questo tipo possono avere comportamenti diversi su sistemi diversi, e l'individuazione delle cause del malfunzionamento sono di difficile individuazione.

Infatti, se il programma di prima viene eseguito su un sistema in cui non si ha l'effetto collaterale dovuto all'*aliasing* fra `a[3]` e `k`, il ciclo termina e l'errore di indicizzazione rimane nascosto.

Occorre quindi prestare *molta attenzione* all'uso degli array e, più in generale, dei puntatori.

Osservazioni

La dimensione di un array deve essere una costante calcolabile dal compilatore (infatti, la dimensione del record di attivazione di una funzione deve essere noto a compile time).

Le seguenti dichiarazioni sono tutte corrette.

```
#define N 10
...
 int b[5*3-5];    // array di 10 elementi di tipo int
 char c[100-2]    // array di 98 elementi di tipo char
 double d[N-2];   // array di  8 elementi di tipo double
```

Nell'ultima riga, il valore di `N` è determinato dal preprocessore.

Non è invece corretta la dichiarazione

```
void f(int n){
  int v[n];
  ...
}
```

in quanto il valore di `n` è noto solo durante l'esecuzione del programma.

Gli array di questo tipo si chiamano *semidinamici*.

Alcuni compilatori (es. `gcc` per Linux) accettano anche dichiarazione di questo tipo.
Con l'opzione `-pedantic` viene segnalato un warning:

```
In function 'f':
warning: ISO C90 forbids variable-size array 'v'
```


Inizializzazione di un array

È possibile inizializzare un array elencando esplicitamente i valori fra parentesi graffe, separati da virgola.

```
int  a[5] = {0, 1, 2, 3, 4};
```

`a[0]` è inizializzato a 0, `a[1]` a 1, ecc.

- Se l'elenco degli inizializzatori è più breve del numero degli elementi dell'array, gli elementi restanti vengono inizializzati a 0.

```
  double v[4] = {2.5 , -3.7};
```

equivale a

```
  double v[4] = {2.5 , -3.7, 0, 0};
```

La definizione

```
  char c[100] = {0};
```

inizializza tutti gli elementi di `c[]` a 0.

- Se non viene dichiarata la dimensione di un array e questo viene inizializzato, viene assunta come lunghezza il numero degli elementi presenti nella inizializzazione.

```
  int c[] = {3, 4, 0};
```

equivale a

```
  int c[3] = {3, 4, 0};
```

Esercizio

```
int v[4] = {10, 20, 30, 40};
int *p[3] = {v+1, v, v+2};
int **q = p+2;
*(q[-2]) = (*q)[-2] + 1;
```

Quali sono i valori nell'array `v[]` dopo aver eseguito le precedenti linee di codice? Verificare la risposta con un programma.

Array come parametri di funzioni

La definizione

```
void f( int a[],  double b[]){
 ...
}
```

è equivalente alla definizione

```
void f( int *a,  double *b){
 ...
}
```

- L'occupazione dei parametri formali `a` e `b` è quella di un puntatore.
- I parametri `a` e `b` vanno inizializzati con *indirizzi* ad array.

L'indirizzo passato non deve essere necessariamente l'indirizzo del primo elemento dell'array, ma può essere l'indirizzo di un *qualunque* elemento dell'array.

Esempio di passaggio di array

Scrivere una funzione `init()` che inizializza un array di interi `a[]` passato come parametro ponendo i suoi elementi uguali a un valore `val` intero passato come parametro.

Soluzione

Anche se non detto esplicitamente, oltre ai parametri `a[]` e `val` occorre un ulteriore parametro `n` che serve a comunicare a `init()` quanti sono gli elementi dell'array che vanno inizializzati.

Il prototipo della funzione è quindi

```
void init( int a[], int n, int val)
```

o, in forma equivalente

```
void init( int *a, int n, int val)
```


Nota

Nel record di attivazione di `init()` non c'è un array, ma un puntatore a un array allocato altrove. L'effetto che si ottiene è che l'array `a[]` è *come se fosse passato per indirizzo*, in quanto le istruzioni del tipo

```
 a[k] = val;
```

modificano l'array passato come parametro.

Codice di `init()`

```
void init(int a[], int n, int val)
/** OPPURE:
 void init(int *a, int n, int val) **/
{
 int k;
  for(k=0; k<n; k++)
    a[k] = val;
}
```


L'istruzione

```
 a[k] = val;
```

può essere sostituita con l'istruzione equivalente

```
 *(a + k) = val;
```

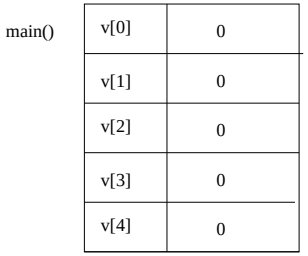
tuttavia la prima notazione è più intuitiva.

Esempio di chiamata

```
int main(){
  int v[5] = {0};
  init(v, 5, 20);
  return 0;
}
```

Esecuzione passo-passo

(1). Viene allocato il record di attivazione di `main()` che contiene l'array `v[]` di 5 elementi inizializzati a 0.



(2). La chiamata

```
init(v, 5, 20);
```

provoca l'allocazione del record di attivazione di `init()`. Il primo argomento passato (di tipo `int*`) è l'indirizzo al primo elemento dell'array `v[]`, e tale valore è posto nel parametro `a` (che è una variabile di tipo `int*`).

Si può anche scrivere

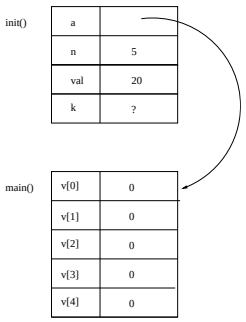
```
init(&v[0], 5, 20);
```

È invece sbagliato scrivere

```
init(&v, 5, 20);
```

perché `v` è già l'indirizzo di `v[0]` (si noti che `&v` ha tipo `int**`, quindi il compilatore segnala un warning).

Dopo il passaggio dei parametri lo stack è:



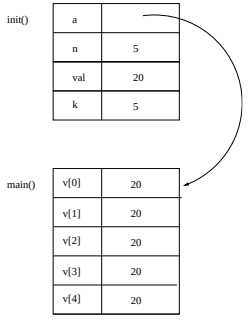
In questa situazione `a[0]` (equivalente a `*(a+0)`, ossia a `*a`) denota la locazione `v[0]`, `a[1]` (equivalente a `*(a+1)`) denota `v[1]`, ecc.

(3). Viene eseguito il ciclo `for`. Alla prima iterazione `k` vale 0, quindi

```
a[k] = val;
```

pone in `v[0]` il valore 20.

(4). Nelle iterazioni successive, vengono modificati i valori di `v[1]`, `v[2]`, `v[3]` e `v[4]`.
Al termine del ciclo `for` lo stack è:



(5). Termina `init()` e l'array `v[]` risulta modificato come richiesto.

Per controllare il risultato, si può usare la seguente funzione `print()` che stampa `n` elementi dell'array `a[]` di `int` passato come parametro.

```
void  print(int a[], int n){
  int k;
  for(k=0; k<n; k++)
    printf("%d ", a[k]);
  putchar('\n');  // a capo
}
```

Quindi, dopo la chiamata a `init()` occorre scrivere l'istruzione

```
print(v, 5);
```

oppure

```
print(&v[0], 5);
```


Altra soluzione

Come si è detto, il parametro `a` di `init()` è un puntatore. Si può allora scrivere la funzione in questo modo: si esegue per `n` volte un ciclo in cui ad ogni iterazione si assegna a `*a` il valore `val` e si incrementa il valore di `a` di uno (nel senso dell'aritmetica dei puntatori).

```
void init(int a[], int n, int val){
  while(n-- > 0) // ciclo ripetuto n volte
    *a++ = val;
}
```


- Il valore di

  `n-- > 0`

  è 1 se il valore (attuale) di `n` è maggiore di 0, altrimenti è 0. Come effetto della valutazione di `n--`, il valore di `n` viene decrementato di 1, quindi il ciclo viene ripetuo `n` volte.
- L'istruzione

  `*a++ = val;`

  è da leggersi come

  `*(a++) = val;`

  ed è equivalente ad eseguire le istruzioni

  ```
  *a = val;
  a++;  //  a = a+1;
  ```

  in quanto il valore di `a++` è il valore attuale di `a` e, come effetto della valutazione, `a` è incrementato di 1.

Nota

L'incremento di `a` non ha nessun effetto sull'array passato come parametro attuale a `init()`!.
Quindi, attenzione a interpretare correttamente l'affermazione *"in C gli array sono passati per indirizzo"*: il significato è che gli assegnamenti fatti ad `a[k]` modificano l'array passato come parametro attuale e non un array "locale" alla funzione chiamata.

Esempio di chiamata

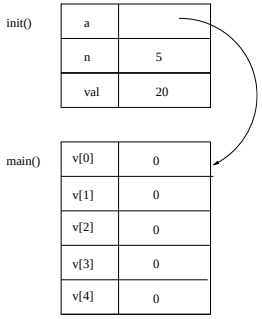
```
int main(){
  int v[5] = {0};
  init(v, 5, 20);
  return 0;
}
```

Esecuzione passo-passo

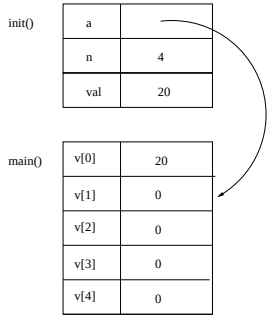
(1). Quando viene eseguita la chiamata

```
init(v, 5, 20);
```

il passaggio dei parametri avviene come prima. Lo stack è:

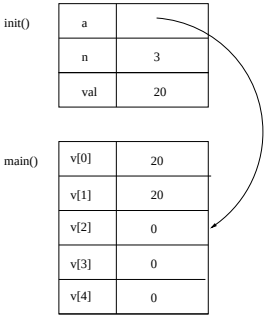


(2). Durante la prima iterazione del ciclo `while` viene posto in `*a` (ossia, in `v[0]`) il valore 20 e la variabile `a` è incrementata di 1. Al termine della prima iterazione lo stack è:



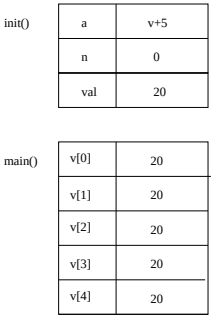
In questo situazione `a[0]` (equivalente a `*(a+0)`, ossia a `*a`) denota la locazione `v[1]`, `a[1]` (equivalente a `*(a+1)`) denota `v[2]`, `a[-1]` (equivalente a `*(a-1)`) denota `v[0]`, ecc.

(3). All'inizio della seconda iterazione, `a` punta a `v[1]`; viene quindi modificato il valore di `v[1]` e, al termine dell'iterazione, lo stack è:



Si noti che la modifica del valore di `a` non ha alcun effetto sull'array `v[]`.

(4). Al termine della quinta iterazione, lo stack è:



Il valore di `a` è l'indirizzo `v+5`, che non corrisponde più ad elementi dell'array `v[]`.
Poiché `n` vale 0, il ciclo termina e anche `init()` termina.
L'effetto prodotto dalla chiamata è quello di prima.

Passaggio di parte di array

È possibile passare a una funzione solamente una parte di un array.

Esempio

Dato l'array

```
int  v[100];  // array di 100 elementi
```

si vuole assegnare il valore -1 agli elementi i cui indici vanno da 20 a 50, estremi compresi, utilizzando la funzione `init()`.

Per ottenere ciò, occorre passare a `init()` come primo argomento l'indirizzo di `v[20]` (primo elemento da inizializzare), come secondo argomento 31 (numero di elementi da inizializzare) e come terzo argomento -1.

La chiamata da eseguire è

```
init(v+20, 31, -1);
```

oppure

```
init(&v[20], 31, -1);
```

Vediamo come viene eseguita la chiamata, considerando le due versioni viste in precedenza.

Prima versione

```
void init(int a[], int n, int val){
 int k;
  for(k=0; k<n; k++)
    a[k] = val;
}
```

Quando `init()` inizia la computazione, la variabile `a` contiene l'indirizzo di `v[20]`. Quando viene eseguita l'istruzione

```
a[k] = val; //  equivale a:  *(a + k) = val;
```

viene modificato il valore nella locazione di indirizzo

```
(v + 20) + k
```

Quindi:

- Quando `k` vale 0, viene modificato `v[20]`.
- Quando `k` vale 1, viene modificato `v[21]`.
- Quando `k` vale 2, viene modificato `v[22]`.
- ...
- Quando `k` vale 30 (ultima iterazione), viene modificato `v[50]`.

Al termine della chiamata il vettore `v[]` risulta modificato come richiesto.

Seconda versione

```
void init(int a[], int n, int val){
  while(n-- > 0)   //  ciclo con n iterazioni
    *a++ = val;
}
```

All'inizio dell'esecuzione, il valore della variabile `a` è l'indirizzo di `v[20]`. Viene quindi eseguite per 31 volte l'istruzione

```
*a++ = val;
```

che modifica la locazione il cui indirizzo è il valore di `a` e ogni volta `a` è incrementato di uno.
Questo significa che:

- Alla prima iterazione viene modificato `v[20]`.
- Alla seconda iterazione viene modificato `v[21]`.
- ...
- Alla 31^a iterazione viene modificato `v[50]`.